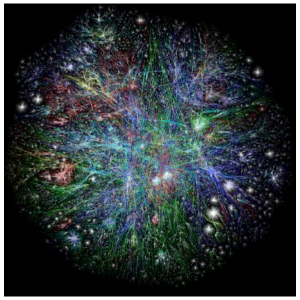
# CS 225

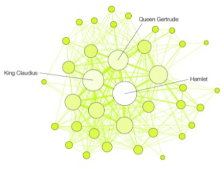
**Data Structures**

## *March 29 – Minimum Spanning Tree (Prim)*

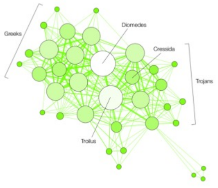
*G Carl Evans*

# Graphs

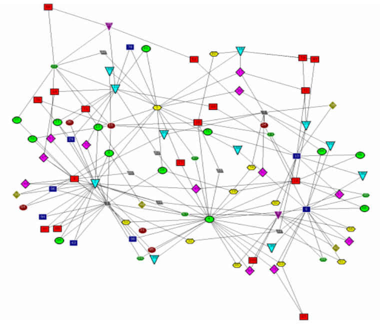




HAMLET

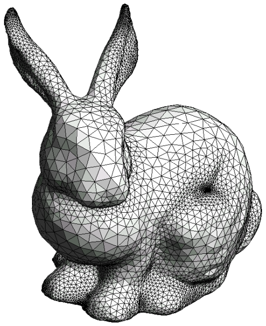


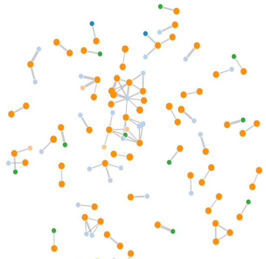
TROILUS AND CRESSIDA

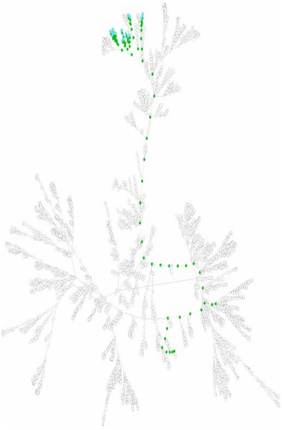


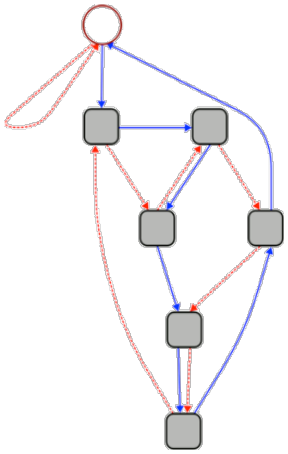
**To study all of these structures:**

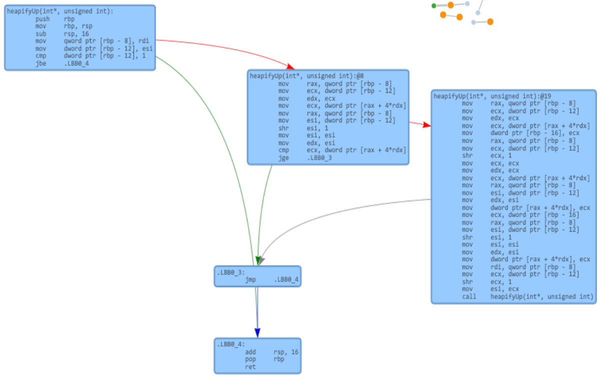
1. A common vocabulary
2. Graph implementations
3. Graph traversals
4. Graph algorithms











```
heapifyUp(int*, unsigned int):
        push    rbp
        mov     rbp, rsp
        sub     rsp, 16
        mov     qword ptr [rbp - 8], rdi
        mov     dword ptr [rbp - 12], esi
        cmp     dword ptr [rbp - 12], 1
        jbe     .LBB0_4
```

```
heapifyUp(int*, unsigned int):@8
        mov     rax, qword ptr [rbp - 8]
        mov     ecx, dword ptr [rbp - 12]
        mov     edx, ecx
        mov     ecx, dword ptr [rax + 4*rdx]
        mov     rax, qword ptr [rbp - 8]
        mov     esi, dword ptr [rbp - 12]
        shr     esi, 1
        mov     esi, esi
        mov     edx, esi
        cmp     ecx, dword ptr [rax + 4*rdx]
        jge     .LBB0_3
```

```
heapifyUp(int*, unsigned int):@19
        mov     rax, qword ptr [rbp - 8]
        mov     ecx, dword ptr [rbp - 12]
        mov     edx, ecx
        mov     ecx, dword ptr [rax + 4*rdx]
        mov     dword ptr [rbp - 16], ecx
        mov     rax, qword ptr [rbp - 8]
        mov     ecx, dword ptr [rbp - 12]
        shr     ecx, 1
        mov     ecx, ecx
        mov     edx, ecx
        mov     ecx, dword ptr [rax + 4*rdx]
        mov     rax, qword ptr [rbp - 8]
        mov     esi, dword ptr [rbp - 12]
        mov     edx, esi
        mov     dword ptr [rax + 4*rdx], ecx
        mov     ecx, dword ptr [rbp - 16]
        mov     rax, qword ptr [rbp - 8]
        mov     esi, dword ptr [rbp - 12]
        shr     esi, 1
        mov     esi, esi
        mov     edx, esi
        mov     dword ptr [rax + 4*rdx], ecx
        mov     rdi, qword ptr [rbp - 8]
        mov     ecx, dword ptr [rbp - 12]
        shr     ecx, 1
        mov     esi, ecx
        call    heapifyUp(int*, unsigned int)
```

```
.LBB0_3:
        jmp     .LBB0_4
```

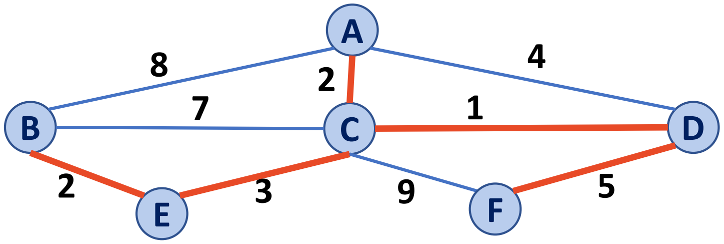
```
.LBB0_4:
        add     rsp, 16
        pop     rbp
        ret
```

# Minimum Spanning Tree Algorithms

**Input:** Connected, undirected graph **G** with edge weights (unconstrained, but must be additive)

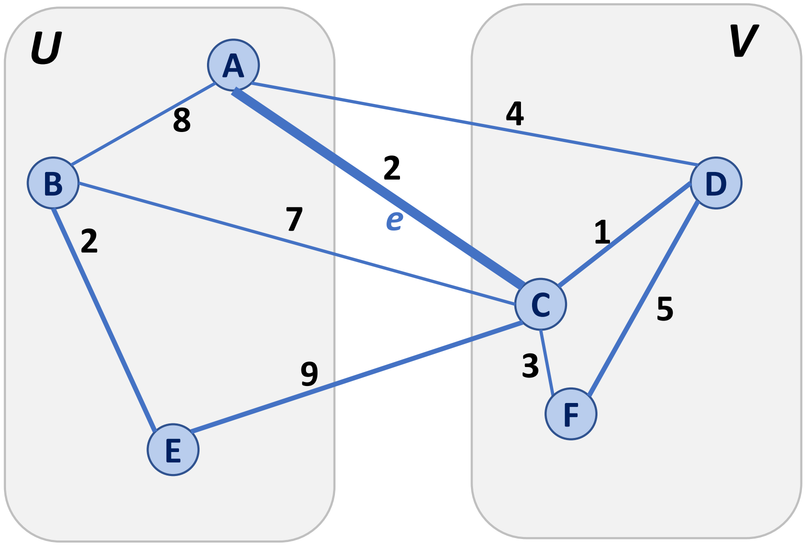
**Output:** A graph G’ with the following properties:

- G’ is a spanning graph of G
- G’ is a tree (connected, acyclic)
- G’ has a minimal total weight among all spanning trees



# Partition Property

Consider an arbitrary partition of the vertices on **G** into two subsets **U** and **V**.

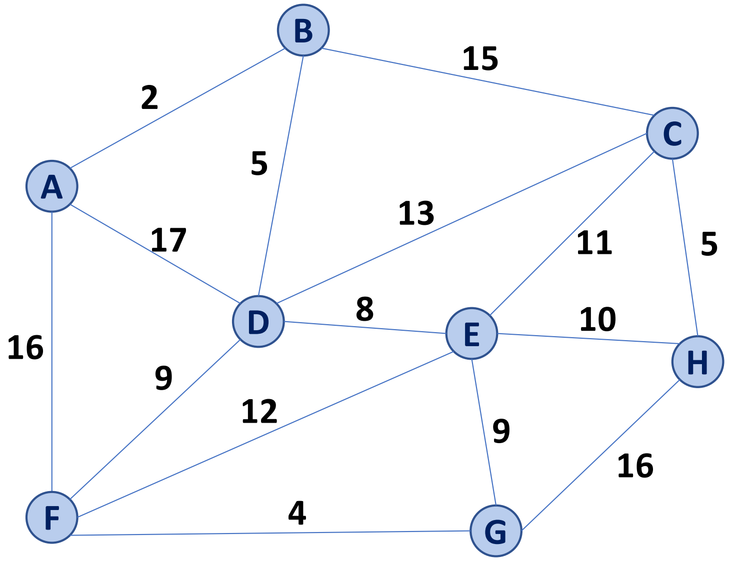


Let **e** be an edge of minimum weight across the partition.

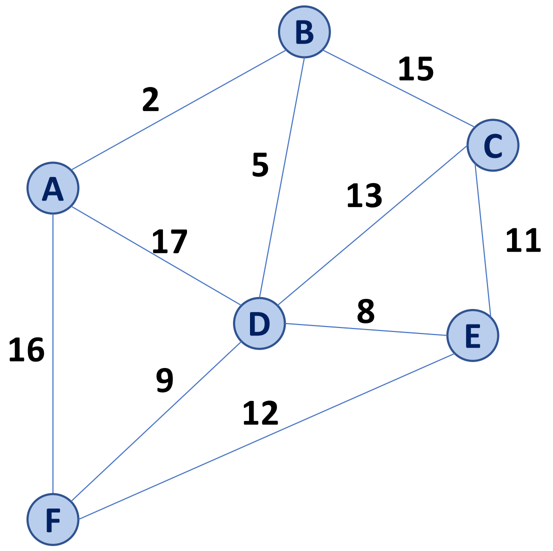
Then **e** is part of some minimum spanning tree.

# Partition Property

The partition property suggests an algorithm:



# Prim’s Algorithm



```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

```
PrimMST(G, s):
  Input: G, Graph;
         s, vertex in G, starting vertex
  Output: T, a minimum spanning tree (MST) of G

  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q   // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m

  return T
```

# Prim’s Algorithm

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

| | Adj. Matrix | Adj. List |
|---|---|---|
| **Heap** | | |
| **Unsorted Array** | | |

# Prim's Algorithm

**Sparse Graph:**

**Dense Graph:**

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T          // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

| | Adj. Matrix | Adj. List |
|---|---|---|
| **Heap** | $O(n \lg(n) + n^2 \lg(n))$ | $O(n \lg(n) + m \lg(n))$ |
| **Unsorted Array** | $O(n^2)$ | $O(n^2)$ |

# MST Algorithm Runtime:

We know that MSTs are always run on a minimally connected graph:

$$n-1 \le m \le n(n-1) / 2$$

$$O(n) \le O(m) \le O(n^2)$$

# MST Algorithm Runtime:

- Kruskal's Algorithm:

  **O(n + m lg(n))**

Sparse Graph:

Dense Graph:

- Prim's Algorithm:

  **O(n lg(n) + m lg(n))**

Sparse Graph:

Dense Graph:

# Suppose I have a new heap:

| | Binary Heap | Fibonacci Heap |
|---|---|---|
| **Remove Min** | O( lg(n) ) | O( lg(n) ) |
| **Decrease Key** | O( lg(n) ) | O(1)* |

**What's the updated running time?**

```
PrimMST(G, s):
  foreach (Vertex v : G):
    d[v] = +inf
    p[v] = NULL
  d[s] = 0

  PriorityQueue Q // min distance, defined by d[v]
  Q.buildHeap(G.vertices())
  Graph T           // "labeled set"

  repeat n times:
    Vertex m = Q.removeMin()
    T.add(m)
    foreach (Vertex v : neighbors of m not in T):
      if cost(v, m) < d[v]:
        d[v] = cost(v, m)
        p[v] = m
```

# MST Algorithm Runtimes:

- Kruskal’s Algorithm:

  **O(m lg(n))**

- Prim’s Algorithm:

  **O(n lg(n) + m lg(n))**

# Shortest Path